###### 3. 操作系统的引导

3.1. 实验目的

熟悉实验环境；

建立对操作系统引导过程的深入认识；

掌握操作系统的基本开发过程；

能对操作系统代码进行简单的控制，揭开操作系统的神秘面纱。

3.2. 实验内容

阅读《Linux内核完全注释》的第 6 章，对计算机和 Linux 0.11 的引导过程进行初步的了解；

按照下面的要求改写 Linux 0.11 的引导程序 bootsect.s ;

有兴趣同学可以做做进入保护模式前的设置程序 setup.s 。

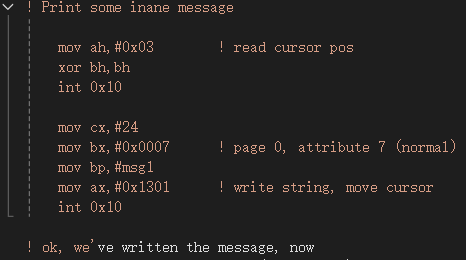
3.2.1. 改写 bootsect.s 主要完成如下功能：

bootsect.s 能在屏幕上打印一段提示信息 XXX is booting...

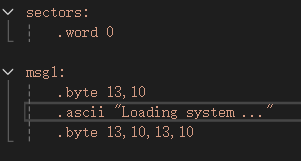
其中 XXX 是你给自己的操作系统起的名字，也可以显示一个特色 logo ，以表示自己操作系统的与众不同。

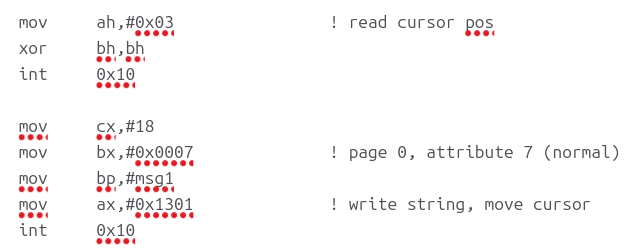
实现过程：

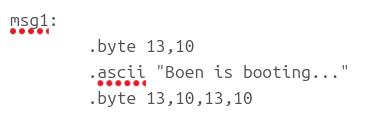
首先先打开这个.s文件，结合指导书上的说明和报告上显而易见的注释可以找到打印部分如下图（为方便阅读遂CV到主机用VS截图）：



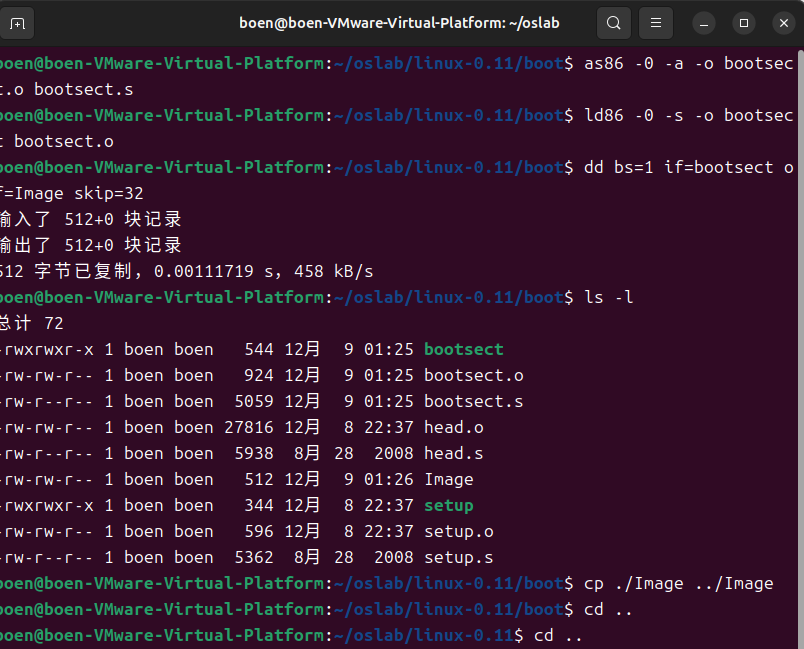
同时，找到MSG1的部分，可以发现打印代码和打印内容：



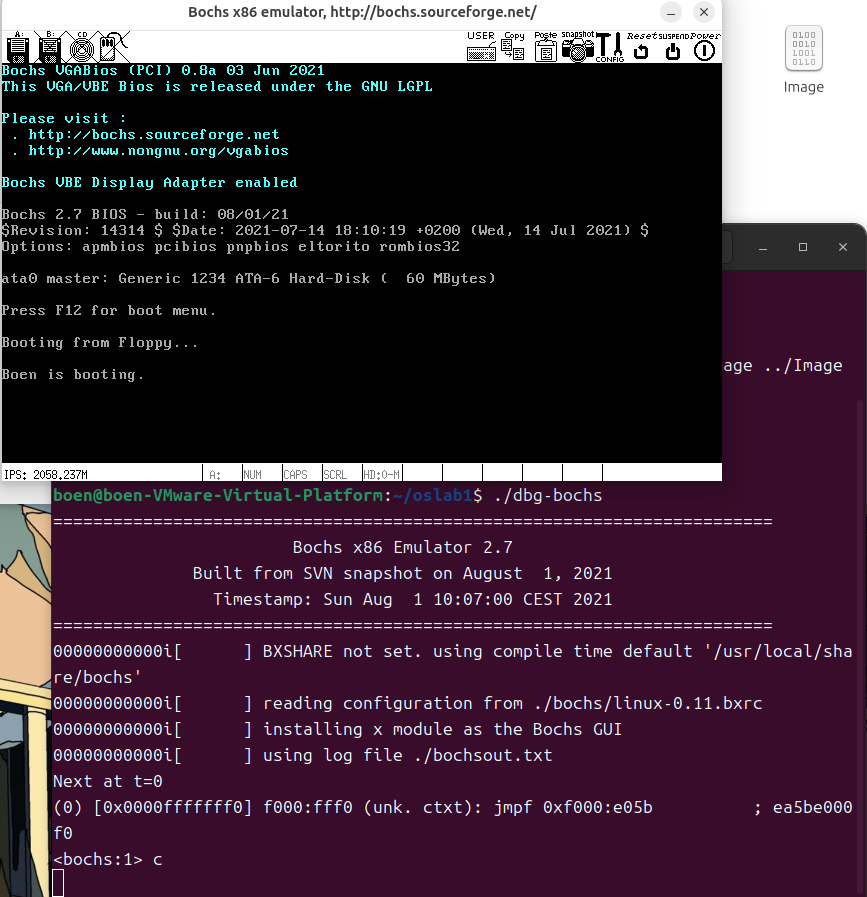
首先先将msg改写成Boen（我的虚拟机的名字） is booting...,显然，消息的长度出现了变化，所以将设置消息长度的部分改为我需要的消息长度：18，最后修改如下：  
 



修改后，对其进行-a -s的编译，再形成一个可执行img文件（由实验指导书引导取出前32字节的文件），将其至于目录下：



再用bochs键入c获得运行界面，得到想要的结果：



3.2.2. 改写 setup.s 主要完成如下功能：

bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行 Now we are in SETUP

setup.s 能获取至少一个基本的硬件参数（如内存参数、显卡参数、硬盘参数等）， 将其存放在内存的特定地址，并输出到屏幕上。

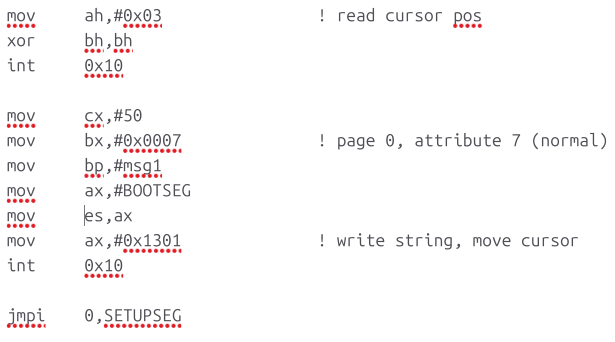
setup.s 不再加载Linux内核，保持上述信息显示在屏幕上即可。

实现过程：

首先完成bootsect.s实现载入setup.s的部分，根据目标和提示，首先将bootsect.s的内容CTRL A+C+V到setup.s中进行修改，将打印语句改为Now we are in SETUP，长度设置为19，并且将跳转部分改为msg2（方便区分），整体修改如下：



之后，修改bootsect.s中打印的部分（以及将扇区改为2）：

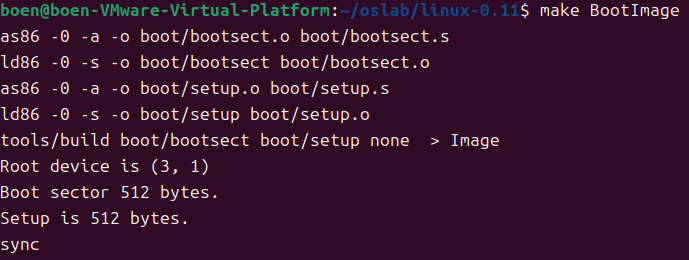


将setup启动定向之后，在print部分添加一段打印完之后跳转的代码，就可以让msg1打印完之后转而去打印msg2了，

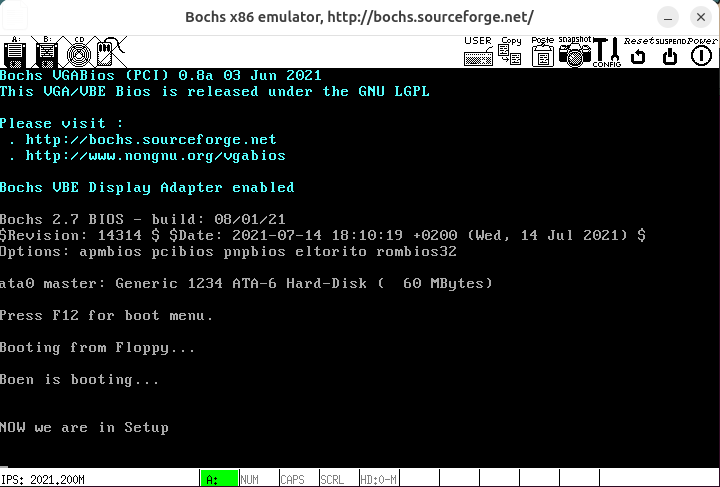
然后，开始处理tools中build.c的部分，第一次使用BootImage时不出意外的报错，观察指导书，很明显发现现在的系统并没有system的部分，索性将其全部注释掉：



之后，按照步骤使用BootImage生成image文件：

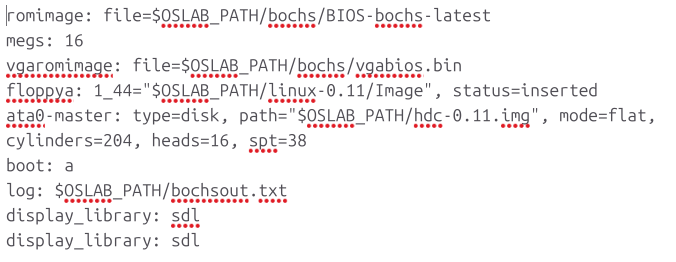


再用bochs运行，成功输出想要的结果：



之后，完成setup打印获取硬件参数的部分，首先，查看bochs中的.bxrc文件了解需要打印的内容，然后，根据刚刚学到的打印方式，找到一个新的setup文件进行缝合，附带检查地址以及其他所需的内容，修改出一个全新的setup：

.bxrc文件所示内容：



新的setup（使用VS进行格式处理）:

INITSEG = 0x9000

    entry \_start

    \_start:

    !屏幕

        mov ah, #0x03

        xor bh, bh

        int 0x10

        mov cx, #25

        mov bx, #0x0007

        mov ax, cs

        mov es, ax  !

        mov ax, #0x1301

        mov bp, #msg2

        int 0x10

        mov ax,#INITSEG

        mov ds,ax

        !读光标位置

        xor bh,bh

        mov ah,#0x03

        int 0x10

        mov [0],dx

        !读扩展内存大小

        mov ah,#0x88

        int 0x15

        mov [2],ax

        mov ax,#0x0000

        mov ds,ax  !中断向量表的起始地址

        lds si,[4\*0x41]  !先存入的是偏移地址，取出存到si中 !取出的4个字节，高位存入ds，低位存入si

        mov ax,#INITSEG

        mov es,ax

        mov di,#0x0004

        mov cx,#16

        rep

        movsb !按字节传送

        mov ax,cs

        mov es,ax

        mov ax,#INITSEG

        mov ds,ax

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#18 !16+2

        mov bx,#0x0007

        mov bp,#msg\_cursor !"Cursor position:" es:bp

        mov ax,#0x1301

        int 0x10

        mov dx,[0]

        call print\_hex

    !显示内存大小

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#14 !12+2

        mov bx,#0x0007

        mov bp,#msg\_memory !"Memory Size:"

        mov ax,#0x1301

        int 0x10

        mov dx,[2]

        call print\_hex

    !补上KB

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#2

        mov bx,#0x0007

        mov bp,#msg\_kb

        mov ax,#0x1301

        int 0x10

    !柱面，cylinder Cyles

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#8

        mov bx,#0x0007

        mov bp,#msg\_cyles

        mov ax,#0x1301

        int 0x10

        mov dx,[4]

        call print\_hex

    !磁头 Heads

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#8

        mov bx,#0x0007

        mov bp,#msg\_heads

        mov ax,#0x1301

        int 0x10

        mov dx,[6]

        call print\_hex

    !扇区 sectors

        mov ah,#0x03

        xor bh,bh

        int 0x10

        mov cx,#10

        mov bx,#0x0007

        mov bp,#msg\_sectors

        mov ax,#0x1301

        int 0x10

        mov dx,[18]

        call print\_hex

    inf\_loop:

        jmp inf\_loop

    !显示硬件参数（16位形式）

    print\_hex:

        mov cx,#4

        !mov dx,(bp)

    print\_digit:

        !使从高位到低位显示4位16进制数

        rol dx,#4

        !ah=0x0e,显示一个字符；al=要显示的字符的ascii码;BIOS中断为int 0x10

        mov ax,#0x0e0f !此时的al未半字节的掩码

        and al,dl !取dl的低4位存入al中

        add al,#0x30

        cmp al,#0x3a !如果是数字，范围是0x30~0x39，即小于0x3a

        jl outp !al小于#0x3a跳转，即数字则跳转

        add al,#0x07 !字母则加上0x07，a~f的范围0x41~0x46

    output:

        int 0x10

        loop print\_digit !每次执行 loop 指令，cx 减 1，然后判断 cx 是否等于 0。 这里即执行4次

        ret

    !打印回车换行

    print\_nl:

        !CR 回车

        mov ax,#0x0e0d

        int 0x10

        !LF 换行

        mov ax,#0x0e0a

        int 0x10

        ret

    msg2:

        .byte 13,10

        .ascii "Now we are in SETUP"

        .byte 13,10,13,10

    msg\_cursor:

         .byte 13,10

         .ascii "Cursor position:"

    msg\_memory:

             .byte 13,10

          .ascii "Memory Size:"

    msg\_cyles:

              .byte 13,10

              .ascii "Cyles:"

    msg\_heads:

              .byte 13,10

              .ascii "Heads:"

    msg\_sectors:

               .byte 13,10

               .ascii "Sectors:"

    msg\_kb:

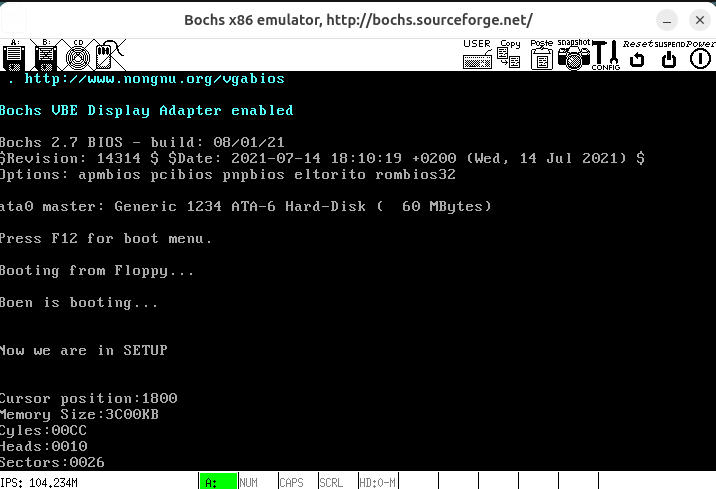
               .ascii "KB"

    .org 508

    boot\_flag:

        .word 0xAA55

打印结果：



结果与原内容相吻合，证明操作成功。

3.3. 实验报告

在实验报告中回答如下问题：

有时，继承传统意味着别手蹩脚。 x86 计算机为了向下兼容，导致启动过程比较复杂。 请找出 x86 计算机启动过程中，被硬件强制，软件必须遵守的两个“多此一举”的步骤（多找

实模式启动：

为什么多此一举：x86 计算机在启动时首先进入实模式，这是为了与古老的 8086/8088 处理器兼容。实模式限制了处理器的寻址能力（只能访问 1MB 内存），并且缺乏保护机制。

替代方案：可以直接从启动开始就进入保护模式。保护模式提供了更完整的内存管理和保护机制，允许操作系统利用更高级的处理器特性。这要求BIOS和操作系统的引导加载器（bootloader）能够直接在保护模式下运行。

BIOS中断调用：

为什么多此一举：在实模式下，操作系统和引导加载器需要通过BIOS中断调用来执行基本输入输出操作，如显示文本、读取磁盘等。这些中断调用是为了兼容早期硬件和软件，但它们效率低下，增加了复杂性。

替代方案：可以直接使用处理器的I/O指令和内存映射I/O（MMIO）来与硬件通信，这样可以省去中断调用的开销，并且使代码更加简洁高效。这同样要求硬件和操作系统支持这些直接操作。

BIOS POST（上电自检）：

为什么多此一举：每次启动时，BIOS都会执行一系列的自检程序来检测硬件配置和状态，这在硬件配置不经常变化的系统中显得有些多余，甚至大部分的笔记本快捷启动时都跳过了这一步骤。

替代方案：可以为系统配置提供一个快速启动选项，假设硬件配置没有变化，从而跳过大部分自检步骤。

MBR（主引导记录）：

为什么多此一举：MBR是一个512字节的扇区，位于磁盘的开始处，它包含了引导加载器的代码和分区表。MBR的设计限制了引导加载器的空间，并且分区表只能容纳四个主分区。

替代方案：使用GPT（GUID分区表）代替MBR，GPT提供了更灵活的磁盘分区机制，并且允许更大的引导加载器空间。

3.3.1. 评分标准

bootsect 显示正确，30%

bootsect 正确读入 setup ，10%

setup 获取硬件参数正确，20%

setup 正确显示硬件参数，20%

实验报告，20%